Storage Efficient Substring Searchable Symmetric Encryption

Iraklis Leontiadis*
Ecole Polytechnique Federale de Lausanne (EPFL)
School of Computer and Communication Sciences
iraklis.leontiadis@epfl.ch

Ming Li
University of Arizona
Department of Electrical and Computer Engineering
lim@email.arizona.edu

ABSTRACT

We address the problem of substring searchable encryption. A single user produces a big stream of data and later on wants to learn the positions in the string that some patterns occur. Although current techniques exploit auxiliary data structures to achieve efficient substring search on the server side, the cost at the user side may be prohibitive. We revisit the work of substring searchable encryption in order to reduce the storage cost of auxiliary data structures. Our solution entails a suffix array based index design, which allows optimal storage cost $O(n)$ with small hidden factor at the size of the string n. Moreover, we implemented our scheme and the state of the art protocol [7] to demonstrate the performance advantage of our solution with precise benchmark results.

CCS CONCEPTS

• **Security and privacy → Database and storage security**; *Security protocols*;

KEYWORDS

substring searchable encryption, suffix array, secure outsourced storage, compression index, privacy

ACM Reference format:
Iraklis Leontiadis and Ming Li. 2018. Storage Efficient Substring Searchable Symmetric Encryption. In *Proceedings of 6th International Workshop on Security in Cloud Computing, Incheon, Republic of Korea, June 4, 2018 (SCC'18),* 15 pages.
https://doi.org/10.1145/3201595.3201598

1 INTRODUCTION

Nowadays, there is a flourish of protocols delegated to run by an untrusted coalition of servers, systems, services, called hereafter the cloud. Due to the untrusted nature of the cloud, users seek to protect the privacy and security of their data with cryptographic primitives. The cloud on the other hand offers an economy of scale with the impressive resources it acquires, ranging from software to hardware. Usually users need to perform a search on their data. Tailored protocols for secure searchable encryption have been proposed in the literature, whereby single or multiple users upload encrypted documents, with some auxiliary data structure called an index, allowing the cloud to correctly return documents containing a single, multiple or a boolean function of keywords, without compromising index, query, and documents privacy.

While keyword based search protocols are quite common in a large range of applications, they cannot efficiently address all the possible queries a user submits to the cloud. Substring based queries have come to the forefront due to the ubiquitousness of devices and the progress in storage technology. Devices produce a big stream of data, which needs to be queried later on with *substring based queries*. Namely a substring query for a stream of data, consists of a substring of the stream and the result is the position of the substring in the big stream, or/and the number of occurrences of multiple substrings.

Applications. In a health-care application, data enclaves which hold giant stream of medical information such as DNA sequencing are asked to answer substring queries by medical labs. The possible position of a substring in the whole DNA sequence of a single person gives information about predisposition to diseases. As such, it is treated as personal sensitive information and should be protected. Nowadays, the sequencing process is possible thanks to the progress of computers. Online services offer DNA sequencing to institutions and individuals. In the logging systems scenario, companies, institutions and organizations produce log data of giant size. The logs are recorded and uploaded in a cloud infrastructure to take advantage of the cheap storage space. Log data are often searched to identify malicious substring patterns. The position of the suspicious searched string token will act as a bookmark to further download the logs data, which proceed and succeed that position for further investigation. Deep packet inspection (DPI) is another application whereby a gateway, firewall, or Intrusion Detection System (IDS) on behalf of a user is looking for prohibitive content on a bigger stream. In general, the vast amount of information renders substring queries a real challenge and reducing the storage cost of the encrypted index would increase the performance of such services.

Protecting the privacy of the data stream and the substring query, while allowing an untrusted cloud to correctly answer substring matching pattern efficiently and securely is not trivial. Following the searchable encryption approach, separating the data itself from the index, results in a prohibitive storage index cost $O(n^2)$, where n is the size of the stream. The index consists of all possible substrings of a stream of data of size n and the encrypted data are the positions of the substring. Recently, the authors in [7] proposed a solution that asymptotically achieves $O(n)$ storage costs by exploiting the auxiliary data structure of the suffix tree. However the asymptotic costs of $O(n)$ hide a constant factor that can be roughly up to 20 [1, 3, 24] for the construction of the suffix tree due to the complexity of the tree and the extra pointers to traverse a tree. Moreover the suffix tree based approach leaks unnecessary information that

*This work was done mostly while the author was affiliated with UofA.

SCC'18, June 4, 2018, Incheon, Republic of Korea

ACM ISBN 978-1-4503-5759-3/18/06…$15.00
https://doi.org/10.1145/3201595.3201598

eventually can reveal all the encrypted positions of the substrings. Following a different approach other than auxiliary index based methods, the authors in [12] achieve to hide the extra leakage at the cost of fixed length substring patterns. The neat of their solution lies on the design of subset sum problems tailored to the positions of specific substrings, so as to the cloud can solve it partially. However this comes at the cost of small constant substring query length during the execution of the protocol.

Our goal, which launches our research is to reduce the increased size of index for substring queries, which has to be computed once and be kept at the cloud during the entire lifetime of the protocol. Even a small improvement would have big impact as the data to be indexed for substring queries span to million of elements. Moreover, besides the cloud side cost cuts, the client will also be positively affected as the smaller the size of the index it is outsourced, the less the charges clients commit to the cloud.

Idea. After encrypting the suffix tree of Chase scheme [7], the encrypted structure leaks a lot of information concerning the internal structure of the tree as number of leaves, children and double "touched" branches. The authors suggested a dummy node policy in order to hide as much information as possible. After constructing the suffix tree with N nodes, the suffix tree is filled up with $2n - N$ internal nodes. To each node with less than σ children, where σ is the size of the vocabulary, up to σ dummy nodes are appended. Encrypting all these dummy blocks drastically increases the storage overhead and subsequently the communication cost of the protocol for index construction.

Our core idea lies at the properties of a suffix array based indexing. A suffix array contains information about the position of each suffix of a string and has constant size n for a string of n size. In contrast, the data structure of the suffix tree has no constant size and can acquire up to $2n$ nodes, with each node storing information about its edges, parent and children nodes, thus increasing the storage need. By choosing the suffix array we decrease the storage need for the construction of the index. The second factor, which allows for less storage and subsequently communication efficiency is the dummy blocks policy which is used to hide the structure of the suffix tree in [7]. Our dummy node policy to obscure the encrypted suffix array relies only on the frequency of the most frequent character. Namely, we fill up the original string with characters such that the frequency of each character is the same. However this approach raises a shortcoming when applied to data sets with skewed frequency distributions such as text based data sets. We overcome that limitation with a bucketization technique. Instead of building the index on a single character approach we explore the idea of grouping together characters consisting a bucket. Surprisingly our experimental evaluation showed that the skewed frequencies are diminished and the final storage overhead for the index is considerably smaller by a factor of **1.8** for text based datasets.

In this paper we design and analyze a storage efficient Substring Searchable Symmetric Encryption (S^3E) protocol with variable size of substrings. We follow a different approach from existing techniques that allows us to achieve the efficiency, functional and security goals we want. In our technique we exploit a self-indexed data structure, which allows the cloud to search for substring queries. Its form resembles the suffix arrays with some additional extra steps.

The main contributions in the paper are summarized as follows:

- *Storage efficient Substring Searchable Symmetric Encryption* (S^3E): Thanks to the employment of the suffix array, which achieves a small hidden factor (≈ 4) in the $O(n)$ asymptotic complexity, compared to the bigger (≈ 20) hidden factor of the suffix tree, our design presents a storage efficient substring searchable symmetric encryption protocol.
- *Variable substring query length*: Our solution allows a dynamic issue of substring queries of variable size without the need of defining a fixed query size beforehand.
- *Provably secure.* Our scheme is provably secure in the real-ideal simulation paradigm with similar leakages as the state of the art scheme in [7].
- *Prototype implementation*: We implemented our protocol and the state of the art work in [7]. We performed a real world comparison of both schemes based on computation time and communication overhead to build the index and query for a substring. Our results show a performance advantage of **1.8** storage overhead on text based datasets as the enron email one.

Outline. In section 2 we introduce the problem this paper addresses. Afterwards, in section 3 we review similar cryptographic protocols for substring searchable symmetric encryption. We continue in section 4 with preliminaries of our solution and the basic building blocks. In section 5 we illustrate in more details the design components of our protocol and in section 6 we describe in details our protocol. We present a thorogh security analysis in Theorem 7. In section 8 we present some benchmark results and in section 9 a comparison with state of the art. Finally, we conclude in section 10.

2 PROBLEM STATEMENT

In this section we formalize the problem of string matching. We first start with the functional requirements of substring matching and afterwards we present the security requirements of the protocol.

2.1 Functional Requirements

Herewith pattern matching, string matching and substring matching are used interchangeably in this paper. We assume that a string S is modeled as a one dimension array $S[1...n]$. A substring is another array $T[1...m]$. The elements of each array are drawn from some finite alphanumerical alphabet Σ of size $\sigma = |\Sigma|$. We say that a substring T occurs in S if there exists $s : 1 \leq s \leq n - m$ and $S[s + 1...s + m] = T[1...m]$, meaning that $S[s + j] = T[j], 1 \leq j \leq m$.

Naive algorithms for pattern matching achieve $O(n)$ on search time and 0 cost on preprocessing. The algorithm simply scans all the positions $i, 1 \leq i \leq n - m$ of the string S until it finds m consecutive matches at a position $j, 1 \leq j \leq n - m + 1$. Trading preprocessing efficiency for better search costs, Robin Karp algorithm [21] achieves $O(n - m + 1)$ search time and $\Theta(m)$ preprocessing amortized cost. In a similar trajectory Knuth-Moris-Pratt [22] has $\Theta(n)$ search complexity and $\Theta(m)$ preprocessing time. Boyer-Moore pattern matching technique [2] increases the preprocessing cost at $\Theta(m + \sigma)$ in order to have worst case search complexity $O(n)$. Following a different trajectory substring matching techniques achieve

$O(m)$ search time by leveraging a more sophisticated preprocessing step, in which the suffixes of all substring are computed along with their positions in the string S, be it suffix tree[25, 34, 35] or suffix array [24]. Suffix tree though has a more expensive space efficiency due to the extra information the suffix tree has to keep [1, 3, 17, 24]. This cost is translated to a constant factor that approximates ≈ 20, which is hidden in the $O(n)$ asymptotic storage cost of the suffix tree construction. As a first step to relax this storage extra hidden cost we choose to build upon the suffix array string matching approach which has a much simpler storage cost which approximates $4n$ [1].

We redraw upon the *queryable encryption* syntactical definition of [7], since we believe it follows a deceptive abstraction. Namely, the functional definition claims to capture a generic framework for searchable encryption, in the sense that a query $\mathcal{F}$ can be any function keyword query, or substring query. However, an encrypted searchable encryption scheme is a more generic protocol, since it can be used to solve the substring searchable encryption problem with the encrypted inverted index technique as shown in the introduction. As such, searchable and substring encryption schemes cannot be addressed by the same definitional framework. Furthermore, the nature of the problem and the solution for substring queries drastically varies from keyword searchable encryption, since the index contains the data and there are not two separate objects, meaning that the index for substring queries is *self-indexed*, since from the index you can recover the underlying data structure. In contrast in encrypted searchable encryption, there is a clear distinction between the index, and the data structure that holds the data (files with keywords). For these reasons we rewrite the functional definitional framework for substring searchable encryption.

Definition 2.1. A Substring Searchable Symmetric Encryption scheme ($\mathsf{S^3E}$) is a collection of four polynomial time algorithms (KeyGen, PreProcess, SrchToken, Search) defined as follows:

- $\mathsf{k} \leftarrow \mathsf{KeyGen}(1^\lambda)$: It is a probabilistic algorithm that takes as input the security parameter in the unary form 1^λ and outputs the secret substring search key k.
- $\mathtt{SES} \leftarrow \mathsf{PreProcess}(\mathsf{k}, S)$: This algorithm takes as inputs the stream S and the secret key k and outputs the substring encrypted data structure SES.
- $\mathsf{tk}_{T,S} \leftarrow \mathsf{SrchToken}(\mathsf{k}, T[1...m])$: It is a probabilistic algorithm that takes as input the secret substring search key k, a string $T[1...m]$ and outputs a trapdoor to search for the string T on data stream S, through SES.
- $(s, \perp) \leftarrow \mathsf{Search}(\mathsf{tk}_{T,S}, \mathtt{SES})$: It is a deterministic algorithm which takes as input a trapdoor $\mathsf{tk}_{T,S}$ and a substring encrypted structure SES and outputs the positions s in S that substring T occurs, or $\perp$ otherwise.

A substring searchable encryption scheme is correct if $\forall \lambda \in \mathbb{N}, \forall S \in \Sigma, \forall \mathsf{k} \leftarrow \mathsf{KeyGen}(1^\lambda), \forall \mathtt{SES} \leftarrow \mathsf{PreProcess}(\mathsf{k}, S), \forall \mathsf{tk}_{T,S} \leftarrow \mathsf{SrchToken}(\mathsf{k}, T[1...m]), \mathsf{Search}(\mathsf{tk}_{T,S}, \mathtt{SES})$ always returns the correct positions s in the string S or $\perp$ otherwise.

2.2 Security Model

Intuitively the security guarantee we ask for is **1**) given a probabilistic polynomial time adversary $\mathcal{A}$ with access to a substring encrypted structure SES, $\mathcal{A}$ cannot gain more partial information about the underlying stream of data S and **2**) given a set of trapdoor tokens for an adaptively generated set of queries $\mathbf{q} = (q_1, q_2, q_3, \ldots, q_o)$ associated with set of tokens $\mathbf{t} = (\mathsf{tk}_1, \mathsf{tk}_2, \mathsf{tk}_3, \ldots, \mathsf{tk}_o)$ $\mathcal{A}$ cannot learn anything for $\mathbf{q}$ and $\mathbf{t}$. Following the symmetric searchable encryption paradigm we know it is impossible to achieve those two security guarantees without leaking some extra information as the observed in [5, 6, 8].

We express the security guarantees of the protocol in terms of simutability [23]. First a leakage function $\mathcal{L}$ is defined, which expresses the leakage of a $\mathsf{S^3E}$ scheme to an adversary $\mathcal{A}$, through the transcripts of the protocol. The simulation framework assumes two worlds. The $\mathbf{Real}^{\mathsf{S^3E}}_{\mathcal{A}(\lambda)}$ world, whereby adversaries can corrupt the parties they want and the $\mathbf{Ideal}^{\mathsf{S^3E}}_{\mathcal{A},\mathcal{S}(\lambda)}$ one in which there is only benign behavior of each party. The security analysis narrows down to the design of a simulator $\mathcal{S}$, who tries to simulate the malicious behavior in the $\mathbf{Ideal}^{\mathsf{S^3E}}_{\mathcal{A},\mathcal{S}(\lambda)}$ world, only through access to the leakage function $\mathcal{L}$. We say that a protocol is secure if $\mathcal{S}$ simulates indistinguishable views of the adversary $\mathcal{A}$ in the $\mathbf{Ideal}^{\mathsf{S^3E}}_{\mathcal{A},\mathcal{S}(\lambda)}$ world.

The adversary $\mathcal{A}$ plays the role of a semi-honest cloud and during the two world we assume a challenger C who interacts with $\mathcal{A}$. We describe the two world in algorithmic details in what is follows:

$\mathbf{Real}^{\mathsf{S^3E}}_{\mathcal{A}(\lambda)}$ world:

- C runs $\mathsf{KeyGen}(1^\lambda)$ to obtain k.
- $\mathcal{A}$ chooses a string $S \in \Sigma$, sends it to C and C replies with $\mathtt{SES} \leftarrow \mathsf{PreProcess}(\mathsf{k}, S)$ to $\mathcal{A}$.
- $\mathcal{A}$ issues a polynomial number of adaptively chosen queries $\mathbf{q} = (q_1, q_2, q_3, \ldots, q_o)$ and receives from C a set of tokens $\mathbf{t} = (\mathsf{tk}_1, \mathsf{tk}_2, \mathsf{tk}_3, \ldots, \mathsf{tk}_o)$.
- Finally $\mathcal{A}$ outputs $v = (\mathtt{SES}, \mathbf{t})$.

$\mathbf{Ideal}^{\mathsf{S^3E}}_{\mathcal{A},\mathcal{S}(\lambda)}$ world:

- $\mathcal{A}$ outputs a string stream S.
- The simulator $\mathcal{S}$ through the leakage $\mathcal{L}$ generates SES and forwards it to $\mathcal{A}$.
- $\mathcal{A}$ issues a polynomial number of queries $\mathbf{q} = (q_1, q_2, q_3, \ldots, q_o)$. $\mathcal{S}$ replies to each of the queries through the leakage function $\mathcal{L}$ with $\mathbf{t} = (\mathsf{tk}_1, \mathsf{tk}_2, \mathsf{tk}_3, \ldots, \mathsf{tk}_o)$.
- Finally $\mathcal{A}$ outputs $v = (\mathtt{SES}, \mathbf{t})$.

Definition 2.2. A substring searchable encryption scheme scheme is adaptively $\mathcal{L}$-semantically secure against a probabilistic polynomial time adversary A if there is exists a polynomial Simulator $\mathcal{S}$ such that for all polynomial time distinguishers $\mathcal{D}$:

$$|\Pr[\mathcal{D}(v) = 1 : v \leftarrow \mathbf{Real}^{\mathsf{S^3E}}_{\mathcal{A}(\lambda)}] -$$
$$\Pr[\mathcal{D}(v) = 1 : v \leftarrow \mathbf{Ideal}^{\mathsf{S^3E}}_{\mathcal{A},\mathcal{S}(\lambda)}]| \leq \mathsf{neg}(\lambda)$$

3 RELATED WORK

Computing on encrypted data goes beyond linear and affine transformations on numerical data held by an untrusted cloud. It is very

common for a single or multiple users to search on encrypted data remotely. Under the scenario of remotely searching, new security guarantees beyond confidentiality and integrity need to be considered. The so called *search pattern* reveals similarities between search queries, and the *access pattern* leaks the identifiers or memory addresses of the accessed files–even encrypted [8].

The ORAM paradigm [16] enables a user to remotely search for encrypted data, without leaking the search or the access pattern. The trade off comes with a bandwidth and communication burden. In [16] the bandwidth overhead is polylogarithmic, which has been reduced down to logarithmic in subsequent work [10, 27, 31–33]. However, in order to provide a real practical real world remote search protocol on encrypted data some leakages are allowed: the *search* and *access* pattern. The formalization of these patterns has been presented in the literature under the **Symmetric Searchable Encryption** (SSE) framework [4–6, 8, 29], with efficient instantiations.

With SSE a user encrypts data and index separately. It uploads both to an untrusted cloud and later on can search efficiently file identifiers with specific single keywords or an expressive boolean function over keywords, without the cloud learning anything about the files or the keywords. This comes at a security cost of leaking the *search* and *access pattern*. Following the approach of SSE, we can design substring searchable symmetric schemes as follows. The user builds an index which maps substrings to positions, encrypts the index and uploads it the the cloud. Later on, the user computes a token for the specific substring and the cloud tries to find a match in the index. If a match occurs the cloud returns the encrypted positions for this token, which correspond to a substring. However, this approach has increased storage cost $O(n^2)$, since the cloud has to keep track of all the possible substrings.

Tailored substring searchable encryptions schemes have been proposed in the literature [7], [12], [11]. Chase *etal.* [7] leverage the auxiliary data structure of the suffix tree. A suffix tree is a compressed suffix trie, can be computed in time $O(n)$ and allows for substring search in O(m) time on a substring of size m. Its amortized storage cost $O(n)$ hides a big constant factor, which could be equal to 20 [1, 3, 17, 24]. In [11] the authors extended the efficient SSE scheme for boolean queries from [4] in order to support substring matching. The idea is to build an index of overlapping k-grams, to prepend its relevant position and encrypt it. When a user needs to perform a substring query, the cloud performs a conjunctive keyword search for all the k-grams of the substring and returns the position. The disadvantage of the scheme comes at the need of storing all the overlapping k-grams at the cloud, which will represent substrings.

In [12] the authors follow a different approach. Instead of taking the *index-then-encrypt* approach with fast symmetric cryptographic primitives, they modify the subset sum problem, which is used to build public key encryption schemes, in a means such that the cloud can solve it. This technique hides also the search pattern but comes at the cost of fixed size substrings, that must be defined in the beginning of the protocol. Moreover the substring should be substantially small with respect to the big stream. Our solution in contrast allows variable size of substring of any size.

Recently, Blass and Moataz [26] strengthen the security requirements by hiding the search and access patterns, following the ORAM approach. By leveraging the Path ORAM technique and the suffix array construction for substring queries, the authors manage to reduce the bandwidth, with a binary recursive tree above the position map. Each node in the tree represents a Path ORAM of the binary search tree for the suffix array. However, in order the cloud to be able to perform an oblivious binary search has to keep track of all the suffixes, which blows up the storage cost for the server. Furthermore the need for storing the suffixes cancels out the suffix array storage advantage over suffix tree. Finally, due to the Path ORAM technique the user has to store a state logarithmic on the length of the string–for the position map. The extra security guarantees of the tailored ORAM scheme do not allow for efficient storage cost both at the client and the cloud side, which is the goal for our work. Sanders *etal.* [30] presented a public key substring searchable encryption, where the underlying substring algorithm is a variant of Rabin-Karp algorithm, thus suffering from linear substring search time. It is noteworthy though to mention that their goal is to decouple keyword generation from the encryption of data in order to allow searches even when the underlying text is not known when the index is built.

Papadopoulos *etal.* [28] addressed the problem of authenticating substring queries without privacy and various work for pattern matching adopts the two party computation model [9, 14, 18, 19] in which one party holds the data stream and a client the pattern. The model differs from the substring searchable symmetric encryption, since in the latter one client holds both the pattern and the stream and uploads an index of the stream to an untrusted party.

4 PRELIMINARIES

In order to reduce the storage cost for our Substring Searchable Symmetric encryption scheme ($\mathsf{S^3E}$) we first substitute the storage expensive suffix tree of the state of art work in [7] with a suffix array SA. A suffix array for a string S of size n constitutes of an integer array of size n, which has at each position a pointer to the start of the matched suffix $T[1...m]$ in the string S. SA is lexicographically sorted with respect to all the possible suffixes and can be computed in linear time on the size of the string S. In order to look for the position of a substring, a binary search in SA is performed, which is used as an index to the original string. Thus, the running time for a substring search is $O(m + logn)$. Let us now consider a concrete example to uncover its details. Suppose S=lalakis. The algorithm for the suffix array proceeds as follows:

(1) Compute all the suffixes starting from the right-most position: s, is, kis, akis, lakis, alakis, lalakis.
(2) Lexicographically sort the suffixes: akis, alakis, is, kis, lakis, lalakis, s.
(3) Find the position in S of each suffix from step 2 and store them in an array $SA = [4, 2, 6, 5, 3, 1]$
(4) Output SA.

However, plugging the SA for a substring searchable symmetric encryption scheme raises some difficulties. We assume that the suffix array is encrypted under a secret key of the user. In order to retrieve the correct encrypted index position from SA, the cloud

should run a binary search obliviously without learning the underlying string S, query substring T, or any of the suffixes. A solution to the problem is to use the technique presented by Gentry *etal.* [15], which allows for a single ORAM query in order to perform a binary search over encrypted data. However, in order to adapt this approach it is required from the server apart from the encrypted suffix array, to store the tree of the encrypted data, which would be an extra burden for its storage complexity.

We take advantage of the *self-indexed* data structure Ferragina-Manzini index, called hereafter FM index [13]. Namely, from FM index the untrusted cloud can answer substring queries by leveraging the suffix array SA, without the need for an ORAM query. The neat property of the FM index is that it can reconstruct the original string S with some extra auxiliary data structures, thanks to its instantiation from the Burrows-Wheeler Transformation algorithm (BWT) [3]. For the reconstruction it employs the LF mapping technique, thus there is no need to store the encrypted stream S. The FM index can be derived from SA, as such its computational overhead is almost for free, after the computation of the suffix array. We describe the core building blocks of the FM index in what it follows.

4.1 Pattern matching

In this section we describe the compressed index FM, that will be used for the construction of our Secure Pattern Matching (S^3E) protocol. The design lies heavily on the BWT transformation for compression of bit-strings and on a special LF mapping for the reconstruction of the original string from BWT. The BWT, along with the LF mapping technique and some auxiliary information are the basic blocks of the compressed index for substring queries.

4.1.1 BWT Transformation. The Burrows-Wheeler Transformation (BWT) transforms a stream of data by leveraging the entropy of each character. In a nutshell, the data stream S is transformed to an encoding W such that compression algorithms provide high rate of compression. For ease of completeness we show the steps to transform an original stream S to W with BWT in algorithm 1. First, the algorithm appends the terminating symbol \$ to the input string S. Then, it builds the matrix W by permuting the symbol \$. At each iteration the permutation is appended as a new row to the matrix W. Finally the rows of W are sorted lexicographically in an ascending way.

4.1.2 LF Mapping. The LF Mapping technique takes the first F and last L columns from the BWT transformation and through an iterative process (algorithm 2) reconstructs the original string S. Starting from the first elements of each column from F and L, the algorithm employees L as an index to the F column. Each time the element of the L column is appended to a LIFO stack. The value at the current position will be used as an index for the F column for the next loop. An example is presented in figure 1. At the first iteration the pointer indicates the first position in both columns F, L. For the next iteration the L character 's' indicates the index for the first column F, which can be found at its last position with F[7] =s. The current character at the L column is appended to a stack D. For the next iteration the current character at the L column indicates the next index for the F column. The character i is pushed to the stack D. The procedure halts when the position at L is \$. Then the algorithm pops all elements from D and the initial string S is fetched.

Algorithm 1: BWT transformation

Input: String S
Output: BWT(S) = W
l=length(S)+1;
S.append(\$);
i=0;
while i<l **do**
  r_i=rotate(s,\$) // *The rotate algorithm permutes the characters of the original string and returns the permuted string*;
  W.addrow(r_i) // *It adds the permuted row from the previous step to the matrix W*;
  i + +;
end
return Sorted.W;

Algorithm 2: LF Mapping

Input: First (F), Last column (L) from BWT
Output: S
D=0 // Initialize the stack D;
l=length.(F) // the length of F equals the length of L;
i=0;
while $L[i]$! = \$ **do**
  D.push(L[i]);
  i=find.F[L[i]]// find.[] denotes the index number in array [] that the element is. For instance find.F['s']=7 ;
while D! = \'0' **do**
  S=S+D.pop;
return S;

F	L	F	L	F	L	F	L	F	L	F	L	F	L	F	L
$	s	$	s	$	s	$	s	$	s	$	s	$	s	$	s
a	l	a	l	a	l	a	l	a	l	a	l	a	l	a	l
a	l	a	l	a	l	a	l	a	l	a	l	a	l	a	l
i	k	i	k	i	k	i	k	i	k	i	k	i	k	i	k
k	a	k	a	k	a	k	a	k	a	k	a	k	a	k	a
l	a	l	a	l	a	l	a	l	a	l	a	l	a	l	a
l	$	l	$	l	$	l	$	l	$	l	$	l	$	l	$
s	i	s	i	s	i	s	i	s	i	s	i	s	i	s	i

Figure 1: The LF mapping process is used to reconstruct the original string S from the transformed one after applying the BWT operation. Starting with the \$ sign from the F column, the mapping progressively reconstructs the entire string S. The last column L is used as a "ladder step" to find the next ith index in the F column, which in turn maps to the ith entry in the L column. The entire procedure halts when L[i]==\$

4.1.3 FM Index. Suffix array vanilla construction has $O(n^2 logn)$ asymptotic computational cost. This stems from the fact that the n suffixes are first sorted by performing $O(nlogn)$ comparisons and each comparison has cost n. Linear time algorithms have been designed by first constructing a suffix tree and then traversing it with a depth first edge in lexicographical order. However, our goal is to be storage efficient, meaning we want to eliminate the storage cost of a suffix tree, which practically approximates a constant factor of $20n$ [1, 3, 17, 24]. We pick up the *skew* algorithm [20] which is a *divide and conquer* based algorithm and achieves linear time construction. The approach of the *skew* algorithm is to recursively divide the suffixes in three groups depending on the position pos of all suffices: pos mod $h, h \in \{1, 2, 3\}$ and then merge the result.

The FM consists of three column arrays. The first one is the F column from the LF mapping, the second one is the L column, which corresponds to the BWT(S) and the last one corresponds to the suffix array SA. SA contains at each row i, the position in the original string S of the substring which corresponds to the i^{th} row of the W matrix obtained after applying the BWT transformation. L = BWT(S) can be computed with the formula BWT(S)[i] = S[SA[i] − 1] from the suffix array. Furthermore for the traversal of the LF mapping the unique ranking of each character in each F, L needs to be stored in r_F, r_L accordingly. Finally $FM = \{F[i], L[i], r_F[i], r_L[i], SA[i]\}_{i=1}^{n}$.

The entire challenge is on how a user encrypts FM in such a way that an untrusted cloud can correctly reply with the encrypted position on substring queries. We give the cryptographic tools that we use in our protocol in the next subsection. We give an intuition of our approach in the next section, we also highlight some shortcoming thereof and we demonstrate our solution to alleviate it.

4.2 Cryptographic Primitives

4.2.1 Pseudorandom functions (PRF). Let the family of all functions in the universe from a domain X to a range Y to be $Func[X, Y]$. A truly random function $f \xleftarrow{\$} Func[X, Y]$ is chosen randomly from the set of $Func$. The set of all these functions is $|Y|^{|X|}$ (gigantic number). It is true that for any random function f with range size L chosen randomly from $Func[X, Y]$, $\Pr[f(x) = y] = 2^{-L}$. The randomness is not parametrized neither by the size of X and Y nor by the size of the domain. We define a pseudorandom function $f_k : X \to Y$ as a function from the set of all functions from X to Y as soon as a particular key k is fixed.

Definition 4.1. Let Func=$\{F : X \to Y\}$ be a function family for all functions F that map elements from the domain X to the range R. Then a $PRF = \{f_k : X' \to Y'\} \subseteq Func$ for $k \xleftarrow{\$} K$, where K is the key space.

The security of a PRF is modeled with a game which is known as *real or random* security game[**?**]. Intuitively, an adversary $\mathcal{A}$ is given access to an oracle that on input x from a domain X, flips a coin $b \xleftarrow{\$} \{0, 1\}$ and if $b = 0$ then it outputs $y = f(x)$, for $f \in Func[X, Y]$, otherwise it outputs $y = f_k(x)$. $\mathcal{A}$ issues queries to the oracle polynomially many times on input of the security parameter λ. Finally $\mathcal{A}$ outputs a guess b' for the bit b.

The advantage of a probabilistic polynomial time algorithm $\mathcal{A}$ in the PRF game is

$$\mathbf{Adv}_{\mathcal{A}}^{PRF} = \Pr[b \xleftarrow{\$} \{0, 1\}; b' \leftarrow \mathcal{A}(y) : b' = b]$$

Definition 4.2. A PRF is computationally secure if all probabilistic polynomially time algorithms $\mathcal{A}$ have advantage in the PRF game: $\frac{1}{2} + \epsilon(\lambda)$, for a negligible function ϵ on the security parameter λ.

4.2.2 Pseudorandom permutations (PRP). A permutation is a bijective function where the domain and the range are equal. Similarly with the random functions, let $Perm[X]$ to be the set of all permutations for the domain X. Then a pseudorandom permutation (PRP) is a randomly chosen permutation from the set $Perm[X]$, keyed under a secret key k.

The advantage of a probabilistic polynomial time algorithm $\mathcal{A}$ in the PRP game is

$$\mathbf{Adv}_{\mathcal{A}}^{PRP} = \Pr[b \xleftarrow{\$} \{0, 1\}; b' \leftarrow \mathcal{A}(y) : b' = b]$$

Definition 4.3. A PRP is computationally secure if all probabilistic polynomially time algorithms $\mathcal{A}$ have advantage in the PRP game: $\frac{1}{2} + \epsilon(\lambda)$, for a negligible function ϵ on the security parameter λ.

4.2.3 Symmetric Key Encryption. A symmetric key encryption scheme SKE = {Gen, Enc, Dec} consists of three algorithms. Gen takes as input a security parameter λ and outputs the secret key sk. The probabilistic encryption algorithm E takes as input the secret key sk and a plaintext x form the plaintext space $\mathcal{P}$ and outputs the ciphertext c. The decryption algorithm SKE.Dec takes as input a ciphertect form the ciphertext space C and the secret decryption key sk and outputs the plaintext $x \in \mathcal{P}$. Correctness follows $\iff \forall \text{sk} \leftarrow \text{Gen}(1^\lambda), \text{SKE.Dec}((\text{E}(\text{sk}, x))) = x, \forall x \in \mathcal{P}$. Security is modeled with the standard game based indistinguishability experiment for polynomial probabilistic time adversary $\mathcal{A}$.

$\underline{\text{PrivK}_{\mathcal{A},\text{SKE}}(\lambda)}$:

- $\mathcal{A}$ has access to the security parameter 1^λ.
- A key $\text{sk} \leftarrow \text{Gen}(1^\lambda)$ is generated and $\mathcal{A}$ can learn encryptions of x of its choice $x \in \mathcal{S} \subset \mathcal{P}$.
- Eventually $\mathcal{A}$ outputs x_0, x_1 where $|x_0| = |x_1|$. $b \xleftarrow{\$}$ and $\text{E}(x_b, \text{sk})$ is returned to $\mathcal{A}$.
- $\mathcal{A}$ outputs its guess for b, b'.

If $b' = b$ $\mathcal{A}$ succeeds and the experiment $\text{PrivK}_{\mathcal{A},\text{SKE}}(\lambda) = 1$.

Definition 4.4. A symmetric encryption scheme SEK has indistinguishable encryptions if the probabilities $\Pr[\text{PrivK}_{\mathcal{A},\text{SKE}}(\lambda) = 1] \leq \frac{1}{2} + \text{neg}(\lambda)$.

5 INTUITION

In this section we provide some intuition about S^3E protocol before delving into its precise description in the follow up section. First we start with showing how the client encrypts the index to allow the cloud process fast encrypted substring queries. Our solution is based on the FM index described in the previous section. The FM index consists of three arrays, which keep track of the F, L columns and the encrypted SA suffix array with positions of substrings and not all the suffixes as in [26]. To recap, the user computes the suffix-array SA and the F, L columns through the BWT transformation.

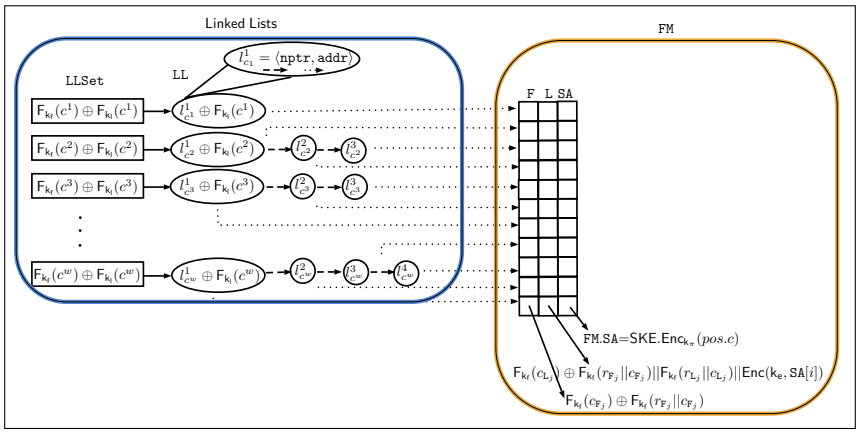

Figure 2: The encrypted FM index construction.

σ	Vocabulary size
Σ	Vocabulary
S	Original stream
T	Substring query
c	character
b	bucket
n	Size of S
m	Size of T
win	window size
SA	Suffix array
F	First Column of LF mapping
L	Last Column of LF mapping
LLSet	Hash map
LL	Linked List
c_{F_j}, $\mathsf{b}_{\mathsf{F}_j}$	j^{th} character,bucket from F column
c_{L_j}, $\mathsf{b}_{\mathsf{L}_j}$	j^{th} character,bucket from L column
r_{c_i}, r_{b_i}	Ranking of i^{th} character,bucket in the string S
r_{F_j}	Ranking of j^{th} character from F column
r_{L_j}	Ranking ofj^{th} character from L column
c^w	w^{th} character, bucket from the alphabet, $(1 \leq w \leq \sigma)$
b^z	z^{th} bucket,$(1 \leq z \leq n - \mathsf{win} + 1)$
λ	Security parameter
$\mathsf{F}(\cdot)$	Pseudorandom function (PRF)
$\Pi(\cdot)$	Psudorandom permutation (PRP)
SKE = {Gen, Enc, Dec}	Symmetric encryption
$\mathsf{k_f}$	PRF key
$\mathsf{k_l}$	PRF key
$\mathsf{k}_{\pi_{1,2,3}}$	PRP keys

Table 1: Notations

5.1 First Approach

We give an overview of our first approach. The protocol can be described in two phases: The encrypted index phase and the search phase.

5.1.1 Encrypted Index. To facilitate the reader we split the encrypted index process in two steps (cf. figure 2) the linked list, which *bootstraps* the search procedure on the FM index by the cloud and 2) FM index itself. The notation used for the protocol is given in table 1. For the security of the scheme the user employs lightweight cryptographic primitives: a pseudorandom function $\mathsf{F}(\cdot)$, a pseudorandom permutation $\Pi(\cdot)$ respectively and a symmetric encryption scheme SKE = {Gen, Enc, Dec}. The untrusted cloud, thanks to the LF mapping and the FM index computation does not need to store all the suffixes of a stream S (cf. figure 2).

Linked List The crux of the design is on how to allow fast indexing through a hash table, which means that there should be unique keys derived from the string with repetitive characters. We employ the ranking information r_c of each character along with the character itself. However, when a user is looking for a substring, it does not know the ranking of each character in the substring $T[1...m]$. We mitigate this deficiency by building a linked list $\mathtt{LL}_c$ for each character.

The user computes a hash table of linked lists LLSet, where each position $\mathtt{LLSet}[\mathsf{F}_{\mathsf{k_f}}(c^w)], 1 \leq w \leq \sigma$ maps to the linked list $\mathtt{LL}_{c^w}$. The number of linked lists equals the number of distinct elements c, denoted as σ in the data stream S, where each symbol $c^w, 1 \leq w \leq \sigma$ comes from an alphabet Σ. The hash table is used to fetch all the positions of a character in the stream S from the linked lists $\mathtt{LL}_{c^w}$. Each linked list $\mathtt{LL}_{c^w}$ stores information concerning the retrieval of the position of c from S. More specifically each node in the list stores the following tuple: $\langle \mathtt{nptr}, \mathtt{addr} \rangle$, nptr is a pointer to the next node of the current list and addr is the address of the element c in the FM index.

The first node of each linked list is stored in the LLSet hash table. In order to prevent the adversary who tries to correlate elements from the LLSet with positions in the FM index, we further encrypt each key $\mathsf{F}_{\mathsf{k_f}}(c^w)$ in the LLSet hash map with another key $\mathsf{k_l}$ as follows: $\mathsf{F}_{\mathsf{k_f}}(c^w) \oplus \mathsf{F}_{\mathsf{k_l}}(c^w)$. Thus the cloud cannot correlate associations from LLSet to FM attack offline without observing any token. The key of the hash map LLSet at $\mathsf{F}_{\mathsf{k_f}}(c^w) \oplus \mathsf{F}_{\mathsf{k_l}}(c^w)$ maps to the first element of the linked list $\mathtt{LL}_c$, which is encrypted as $\langle \mathtt{nptr}, \mathtt{addr} \rangle \oplus \mathsf{F}_{\mathsf{k_l}}(c^w)$[1]. As such, the frequency of each character before a search query is hidden.

However, once the cloud receives queries, it can learn the frequency of encrypted characters in the linked list which represent characters of the string. In order to obfuscate frequency analysis on the encrypted index from substring search queries, we pad the data stream with dummy blocks. These dummy blocks make all linked

[1]Notice that even if we use a one time pad with the same key for two different elements: $a = \langle \mathtt{nptr}, \mathtt{addr} \rangle$, $b = \mathsf{F}_{\mathsf{k_f}}(c^w)$ an adversary by xoring the two ciphertexts encrypted under the same key $\mathsf{F}_{\mathsf{k_l}}(c^w)$, learns $ab = \langle \mathtt{nptr}, \mathtt{addr} \rangle \oplus \mathsf{F}_{\mathsf{k_f}}(c^w)$, which is a one time pad encryption of $\langle \mathtt{nptr}, \mathtt{addr} \rangle$ with key $\mathsf{F}_{\mathsf{k_f}}(c^w)$.

lists to appear with the same size. The core idea for padding is to produce dummy blocks from the vocabulary Σ depending on the ranking of the most frequent character. E.g: Original stream=abbcd and Σ=abcd, then the dummy blocks equal dc={ a, c, d}. Finally the user chooses uniformly at random $dpos \xleftarrow{\$} \{|dc|\}$ and appends the original string S at position $dpos$ of dc. Following the previous example; if dpos=1 then S'=aacdbbcd. The cloud responds in the final round with the encrypted position pos. User accepts the result as correct if $dpos \leq pos \leq n - dpos$ and $pos + m < dpos + n$.

FM Index Encryption (figure 2). The second difficulty comes when the cloud tries to traverse the FM index through the LF mapping technique. The encrypted FM index contains unique digests of characters, while the cloud should identify matches from the token $\mathsf{tk}_{T,S}$, that encodes repetitive characters deterministically. In order to allow the cloud traverse the encrypted FM index, we encrypt the FM as a key value hash table where the key consists of $\mathsf{F}_{\mathsf{k_f}}(c_{\mathsf{F}_j}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{F}_j}||c_{\mathsf{F}_j})$ and the value is $\mathsf{F}_{\mathsf{k_f}}(c_{\mathsf{L}_j}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{F}_j}||c_{\mathsf{F}_j})||\mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{L}c_j}||c_{\mathsf{L}_j}), \mathsf{E}(\mathrm{pos}_j)$. Finally the user permutes all the tuples with a secure permutation: $\Pi_{\mathsf{k}_\pi}(t_j)$.

5.1.2 Search. During the Search phase on a substring query $\mathsf{tk}_{T,S} = \mathsf{F}_{\mathsf{k_f}}(T[1...m]) = \mathsf{F}_{\mathsf{k_f}}(T[1]), \mathsf{F}_{\mathsf{k_f}}(T[2]), \ldots \mathsf{F}_{\mathsf{k_f}}(T[m]), \mathsf{F}_{\mathsf{k_l}}(T[m])$ the cloud proceeds as follows:

Bootstrap. First it needs to *bootstrap* the search by finding the correct candidate positions in the encrypted FM table through the linked list, which correspond to all positions in the string S where the last character of the query possibly exists. From the LLSet hash table it looks for the value with key $\mathsf{F}_{\mathsf{k_f}}(T[m]) \oplus \mathsf{F}_{\mathsf{k_l}}(T[m])$. This value maps to a linked list $\mathtt{LL}_c$, in which each node maps to the encrypted FM tuple $t_j = t_j^0, t_j^1, t_j^2 =$

$$\langle \underbrace{\mathsf{F}_{\mathsf{k_f}}(c_{\mathsf{F}_j}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{F}_j}||c_{\mathsf{F}_j})}_{\mathsf{F}},$$

$$\underbrace{\mathsf{F}_{\mathsf{k_f}}(c_{\mathsf{L}_j}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{F}_j}||c_{\mathsf{F}_j})||\mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{L}c_j}||c_{\mathsf{L}_j})}_{\mathsf{L}}, \underbrace{\mathsf{E}(\mathrm{pos}_j)}_{\mathsf{SA}}\rangle_{j=1}^{n}.$$

In order to decrypt the first element of the linked list the cloud uses $\mathsf{F}_{\mathsf{k_l}}(T[m])$ as a key to decrypt $\langle \mathtt{nptr}, \mathtt{addr}\rangle \oplus \mathsf{F}_{\mathsf{k_l}}(c^w)$ and learns $\langle \mathtt{nptr}, \mathtt{addr}\rangle$. The cloud uses $\mathsf{F}_{\mathsf{k_f}}(T[m])$ and applies a xor operation on the F column at the ranges that it retrieved from the linked list of the c_m character $\mathtt{LL}_c$ and learns $\mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{F}_j}||c_{\mathsf{F}_j})$.

Iteration. The cloud uses $\mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{F}_j}||c_{\mathsf{F}_j})$ as a key to decrypt the first part of the L column element $\mathsf{F}_{\mathsf{k_f}}(c_{\mathsf{L}_j}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{F}_j}||c_{\mathsf{F}_j})$ and reveals $\mathsf{F}_{\mathsf{k_f}}(c_{\mathsf{L}_j})$. It then fetches the encrypted L column as $k = \mathsf{F}_{\mathsf{k_f}}(c_{\mathsf{L}_j}), b = \mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{L}_j}||c_{\mathsf{L}_j})$ in which $\mathsf{F}_{\mathsf{k_f}}(c_{\mathsf{L}_j}) = \mathsf{F}_{\mathsf{k_f}}(T[m-1])$ and for all nodes from the linked list computes $k \oplus b$, which is used as a key for the F column. The procedure terminates when the processed substring character is the first one $\mathsf{F}_{\mathsf{k_f}}(T[1])$.

At this point the cloud returns to the user all the encrypted $\mathsf{E}(\mathrm{pos}_j)$ for the substrings. The user decrypts and accepts the result as long as the decrypted position is in the range of the size of original stream without padding.

Second round. From the per-character one way function $\mathsf{F}_{\mathsf{k_f}}$ evaluation of the substring query: $\mathsf{tk}_{T,S} = C[1], C[2], \cdots, C[m] \leftarrow \mathsf{F}_{\mathsf{k_f}}(T[1...m])$ and the LF mapping the protocol leaks to the cloud in cleartext the exact differences of the positions of two encrypted substring in the stream S as long there are unique matches. More specifically, the number of iterations in the LF mapping traversal (algorithm 2), reveals how many positions two substrings they differ, as long as there is *unique* match in S. Eventually, an untrusted cloud can decrypt the entire encrypted SA array, which contains encrypted positions of all substrings in S, since it knows its addresses. To circumvent the leakage we first use two different permutations to permute the tuples t_j: $\Pi_{\mathsf{k}_{\pi_1}}(t_j^0, t_j^1) = \pi_j^{0,1}, \Pi_{\mathsf{k}_{\pi_2}}(t_j^2) = \pi_j^2$. As such, after the permutation $\underbrace{\mathsf{F}_{\mathsf{k_f}}(c_{\mathsf{F}_j}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{F}_j}||c_{\mathsf{F}_j})}_{\mathsf{F}}, \underbrace{\mathsf{F}_{\mathsf{k_f}}(c_{\mathsf{L}_j}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{F}_j}||c_{\mathsf{F}_j})||\mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{L}c_j}||c_{\mathsf{L}_j})}_{\mathsf{L}}$ are stored in position $\pi_j^{0,1}$ at the FM array and $\underbrace{\mathsf{E}(\mathrm{pos}_j)}_{\mathsf{SA}}$ at position π_j^2. The cloud as traverses the token returns the permuted encrypted position of the substring token, the client applies the inverse permutation and fetches the correct cell from the FM array. By doing so the cloud cannot learn on its own, the encrypted position of a substring. The second permutation prevents him to learn this information by stopping the traversal of the index at any substring of the original query at its choice. To perform that, it needs the contribution of the user. In part, the second permutation can be viewed as the *induced permutations* of *structured encryption* [6], which encrypt positions of items belonging a data structure, but in contrast with *structured encryption* a user in substring queries is interested to identify positions of sub-elements (substrings) of the original data sructure and not the entire elements (e.g: positions of elements in a matrix).

5.2 Improved scheme

5.2.1 Shortcoming. Recall that during the encryption of the index, the user adds dummy blocks at the linked lists in order to alleviate frequency attacks. Namely after issuing a query token $\mathsf{tk}_{T,S}$ the client reveals to the cloud the key $\mathsf{F}_{\mathsf{k_l}}(T[m])$ in order to locate the head of the list which corresponds to the key $\mathsf{F}_{\mathsf{k_f}}(T[m]) \oplus \mathsf{F}_{\mathsf{k_l}}(T[m])$. The cardinality of the list corresponds to the frequency of that character in the original string S. In our first approach we address this problem by adding dummy blocks in all linked lists in order their size to equal the size of the longest one.

The aforementioned technique protects the client from frequency attacks on the original string and imposes low overhead in case of a string drawn from a distribution with homogeneous frequency characters. However in a more *skewed* dataset with characters having broad frequencies, then the technique of dummy characters can drastically affect the efficiency of the system. More specifically the dummy characters may double the size of the final size of the index, thus degrading the storage overhead and the computational efficiency of the client.

5.2.2 Bucketization of characters. **Index.** The entire procedure is similar to the first approach but instead of operating on single characters everything operates on buckets. Now the F and L column arrays correspond to buckets and the input to the $\mathsf{F}_{\mathsf{k_f}}$ is not a single character c but a bucket b. We use the same notation conventions with the first approach but instead of operating on characters we operate on buckets, i.e: we denote by b^1 the first bucket of string,

$\mathsf{b}_{\mathsf{F}_j}$ is the j^{th} bucket of the F column, etc. The LLSet hashmap stores for each key $\mathsf{F}_{\mathsf{k_f}}(\mathsf{b}^z) \oplus \mathsf{F}_{\mathsf{k_l}}(\mathsf{b}^z)$, the head of lists, which correspond to buckets of the stream. The entire procedure to encrypt the index is identical with our basic approach and we omit a repetitive overview thereof.

Search. During the search phase if the size of the token, which consists of possible consecutive buckets of the original string S, is a multiple of the windows size win then the protocol protocol proceeds identically as with the *per-character* previous version of the search procedure.

However when the size of the token is not multiple length of win then there will be always a faulty mismatch. The possible match of the last bucket of the token will be inside the last non-matched bucket from the index. Recall that from the LF traversal on the FM index, the search starts from the last character-bucket and proceeds up to the first character-bucket of the search token. As such the cloud during the search phase it will always misfire a mismatch. We alleviate the correctness problem as follows. The cloud starts the search not from the last bucket of the token $\mathsf{F}_{\mathsf{k_f}}(T[m])$ but for the previous one $\mathsf{F}_{\mathsf{k_f}}(T[m-1])$. If there is a match for all $m-1$ blocks of the token then the client needs to decrypt the mth bucket in the string to verify matching of the first $e = n \bmod \mathsf{win}$ remaining characters which correspond to the last bucket of the query token. To do so the client has to encrypt and upload the stream S similarly with [7] in a per bucket fashion resulting to an array of encrypted buckets B. To avoid the leakage of the position the client permutes the buckets with a pseudorandom permutation Π, keyed by k_{π_3} : $B'[j] = B[\Pi_{\mathsf{k}_{\pi_3}}(j)]$ and instead of querying for the mth bucket it forwards a request for bucket number $B[\Pi_{\mathsf{k}_{\pi_3}}(m)]$. It then decrypts the bucket and compares it with the last bucket from the query to identify a matching happening at position m.

6 PROTOCOL

We give the full details of our substring searchable symmetric encryption protocol, which alleviates the storage overhead shortcoming of the first approach with our bucketization technique:

- $\mathsf{k} \leftarrow \mathsf{KeyGen}(1^\lambda)$: This algorithm runs by the user takes as input the security parameter 1^λ and generates random keys $\mathsf{k} = (\mathsf{k_f}, \mathsf{k_l}, \mathsf{k}_{\pi_{1,2,3}}, \mathsf{k_{e,c}})$ for a PRF $\mathsf{F}_{\mathsf{k_f}} : \{0,1\}^\lambda \times \{0,1\}^\nu \rightarrow \{0,1\}^\mu$, a PRP $\Pi_{\mathsf{k}_\pi} : \{0,1\}^\lambda \times \{0,1\}^\nu \rightarrow \{0,1\}^\nu$ and a symmetric encryption algorithm SKE = {Gen, Enc, Dec}. Finally it outputs k to the user. For the generation of the keys we assume a source of randomness $\mathcal{R}$ and a pseudorandom generator G seeded with

$$s_\mathsf{f} \xleftarrow{\$} \mathcal{R}, s_\mathsf{l} \xleftarrow{\$} \mathcal{R}, s_{\pi_{1,2,3}} \xleftarrow{\$} \mathcal{R}, s_\mathsf{e} \xleftarrow{\$} \mathcal{R} :$$
$$(\mathsf{k_f}, \mathsf{k_l}, \mathsf{k}_{\pi_{1,2,3}}, \mathsf{k_{e,c}}) \leftarrow \mathsf{G}(s_\mathsf{f}), \mathsf{G}(s_\mathsf{l}), \mathsf{G}(s_{\pi_1}), \mathsf{G}(s_{\pi_2}), \mathsf{G}(s_\mathsf{e})$$

- $\mathsf{SES} \leftarrow \mathsf{PreProcess}(\mathsf{k}, S)$: User owns a stream S, which contains characters $c \in \Sigma$. S has n characters. Let $\mathsf{max_b}$ be the cardinality of most frequent bucket and f_{b_i} the frequency of bucket b_i. User:
  (1) Parses the string S as buckets: $\{\mathsf{b}_z\}_{z=1}^{n-\mathsf{win}+1}$, each of size win characters and k in total distinct buckets:
  for $z = 1; z++; z = n - \mathsf{win}$
  $\mathsf{b}_z = (z + \mathsf{win} \leq |S|)?S[z...z + \mathsf{win}] : S[z...n - z]$
  (2) Chooses dummy buckets $dc = \sum_{j=1}^{k} \mathsf{max_b} - f_{\mathsf{b}_i}$ that constitute a dummy stream. The user chooses uniformly at random $dpos \xleftarrow{\$} \{|dc|\}$ and appends the original string S at position $dpos$ of dc
  (3) Computes the suffix array SA and the F, L columns on input the buckets B and stores them as the FM index: FM = F||L, SA.
  (4) Encrypts all buckest $B = \mathsf{SKE.E}(\mathsf{k_c}, \mathsf{b}_z), 1 \leq z \leq n + k - \mathsf{win} + 1$
  (5) Permutes $B'[j] = B[\Pi_{\mathsf{k}_{\pi_3}}(j)]$
  (6) Encrypts elements of SA array with $\mathsf{SKE.E}(\mathsf{k_e}, \mathsf{SA}[i]), 1 \leq i \leq +k - \mathsf{win} + 1$.
  (7) Applies the PRF to each element of F as follows:
  $$\mathsf{F}[i] = \mathsf{F}_{\mathsf{k_f}}(\mathsf{b}_{\mathsf{F}_i}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{F}_i}||\mathsf{b}_{\mathsf{F}_i})$$
  (8) Computes
  $$\mathsf{L}[i] = \mathsf{F}_{\mathsf{k_f}}(\mathsf{b}_{\mathsf{L}_i}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{F}_i}||\mathsf{b}_{\mathsf{F}_i})||\mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{L}_i}||\mathsf{b}_{\mathsf{L}_i})$$
  (9) Applies a pseudorandom permutation Π_{k_π} to the tuples:
  $$t_0 = \mathsf{F}_{\mathsf{k_f}}(\mathsf{b}_{\mathsf{F}_i}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{F}_i}||\mathsf{b}_{\mathsf{F}_i}),$$
  $$t_1 = \mathsf{F}_{\mathsf{k_f}}(\mathsf{b}_{\mathsf{L}_i}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{F}_i}||\mathsf{b}_{\mathsf{F}_i})||\mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{L}_i}||\mathsf{b}_{\mathsf{L}_i})$$
  using k_{π_1} and with k_{π_2} user permutes $\mathsf{E}(\mathsf{k_e}, \mathsf{SA}[i])$, for $i = 1, ..., n + k - \mathsf{win} + 1 : \Pi_{\mathsf{k}_{\pi_1}}(t_i^0, t_i^1) = \pi_i^{0,1}, \Pi_{\mathsf{k}_{\pi_2}}(t_i^2) = \pi_i^2 = \mathsf{FM}'$.
  (10) For every distinct bucket in $\mathsf{F}[i] = \mathsf{F}_{\mathsf{k_f}}(\mathsf{b}_{\mathsf{F}_i}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathsf{F}_i}||\mathsf{b}_{\mathsf{F}_i})$ the user initiates a linked list LL_c and at each node stores $\mathsf{LL}_c.\mathsf{nptr}$ for the next node of the list and $\mathsf{LL}_c.\mathsf{addr}$ which points to the tuple t_i with a matching $\mathsf{F}_{\mathsf{k_f}}(\mathsf{b}_{\mathsf{F}_i})$. Finally it encrypts the first element of each linked list LL_c with $\mathsf{F}_{\mathsf{k_l}}(\mathsf{b}_i) : \langle \mathsf{nptr}, \mathsf{addr} \rangle \oplus \mathsf{F}_{\mathsf{k_l}}(\mathsf{b}_i)$.
  (11) Stores the head pointers of the collections of all linked lists in a hash table LLSet with key $k = \mathsf{F}_{\mathsf{k_f}}(\mathsf{b}_i) \oplus \mathsf{F}_{\mathsf{k_l}}(\mathsf{b}_i)$ and value v a pointer to the head of the list LL_c, which stores information about the $\mathsf{F}_{\mathsf{k_f}}(\mathsf{b}_i)$ character, meaning all its positions to the encrypted FM index.
  (12) Finally outputs $\mathsf{SES} = (\mathsf{LLSet}, \mathsf{LL}_c, \mathsf{FM}', B')$ and keeps only the keys $\mathsf{k} = (\mathsf{k_f}, \mathsf{k_l}, \mathsf{k}_{\pi_{1,2,3}}, \mathsf{k_{e,c}})$.
- $\mathsf{tk}_{T,S} \leftarrow \mathsf{SrchToken}(\mathsf{k}, T[1...m])$: This algorithm takes as input the secret substring search key k, a string $T[1...m]$ and outputs a trapdoor to search for the string T on data stream S, through SES:
  (1) Parse $T[1...m]$ to buckets $T_b = \{T_b^i\}_{i=1}^{m-\mathsf{win}}$ of size win:
  for $i = 1; i++; i = m - \mathsf{win}$
  $T_b^i = (i + \mathsf{win} \leq |S|)?T[i...i + \mathsf{win}] : T[i...m - i]$
  (2) User with his secret PRF key $\mathsf{k_f}$ computes $\mathsf{tk}_{T,S} = C[1], C[2], \cdots, C[m - \mathsf{win}] \leftarrow \mathsf{F}_{\mathsf{k_f}}(T_b = \{T_b^i\}_{i=1}^{m-\mathsf{win}-1}), \mathsf{F}_{\mathsf{k_l}}(T_b^{m-\mathsf{win}})$ and forwards $\mathsf{tk}_{T,S}$ to the cloud.
- $(s, \perp) \leftarrow \mathsf{Search}(\mathsf{tk}_{T,S}, \mathsf{SES})$: The cloud parses the token query $\mathsf{tk}_{T,S} = C[1], C[2], \cdots, C[m - \mathsf{win}]$ and searches the position in S from the encrypted index SES as follows:
  (1) **if** $|\mathsf{tk}_{T,S}| \bmod \mathsf{win} == 0$
  $x = m - \mathsf{win}, \mathsf{flag} = 0$
  else

$x = m - \text{win} - 1, \text{flag} = 1$ // Search for the last equal size bucket pattern before the last one.

(2) $u = find(\text{LLSet}, C[x])$//find in dictionary LLSet the value u with key $C[x]$.u is a pointer to the head of a list LL, which stores pointers to all buckets $T[m]$ in S

(3) **if** $u == \perp$ **return** $\perp$

(4) **while** $u \neq \perp$ **do**

$K = K \cup u.\text{addr}$ //traverse the list and store in the set K the addresses of the characters.

$u = u.\text{nptr}$

(5) **for** $p = x - 1; p > 1; p = p - 2$

for $i = 1; i < size(K); i++$

if $\text{SES.L}[K[i])^{(1)} == C[p-1]$//Store in the set $KEYS$ only the elements from the F column, whose associated L element equals the next bucket from C in a backword order. $\text{SES.L}[K[i]]^{(1)}$ maps to $\text{F}_{\text{k}_\text{f}}(\text{b}_{\text{L}_i}) \oplus \text{F}_{\text{k}_\text{f}}(r_{\text{F}_i}||\text{b}_{\text{F}_i})$ and $\text{SES.L}[K[i]]^{(2)}$ to $\text{F}_{\text{k}_\text{f}}(r_{\text{L}_i})$.

$KEYS = KEYS \cup \text{SES.L}[K[i]]$

else $K = K - K[i]$ //Remove all the non matched elements from the key set K.

if $K == \perp$ **return** $\perp$

for $i = 1; i < size(KEYS); i++$

$r_{\text{F}_i} = C[p] \oplus KEYS^{(1)}[i]$

$z = r_{\text{F}_i} \oplus KEYS^{(2)}[i]$ //Compute the key from the L column as $KEYS^{(1)}[i] \oplus KEYS^{(2)}[i]$, which corresponds to $\text{F}_{\text{k}_\text{f}}(\text{b}_{\text{F}_i}) \oplus \text{F}_{\text{k}_\text{f}}(r_{\text{F}_i}||\text{b}_{\text{F}_i})$ in the F column of the SES object.

if $\text{SES.F}[z] \neq \perp$

continue

else $KEYS = KEYS - KEYS[i]$ //Remove all the non matched elements from the key set $KEYS$.

$K = KEYS$

(6) **if** $K == \perp$ **return** $\perp$

(7) The cloud sends to the user $\text{SES.FM}'[K]$. The client runs the inverse permutation Π_{π_2} to the K indexes and gets back $\{i'\}$ and asks the cloud for $\text{SES.FM}'[\{i'\}]$. After getting back the results the client decrypts $\text{pos} = \text{E}(\text{k}_\text{e}, \text{SA}[\{i'\}])$ with k_e and learns the position pos of the asked substring T in S. In case flag == 0 user accepts the result as correct if $dpos \leq pos \leq n - dpos$ and $\text{pos} + m < dpos + n$.

(8) **if** flag == 1 recap that the LF traversal on the FM index starts at the last character and proceeds invertly. Notice that we excluded at step **1** during the **else** branch the remaining last bucket as it will always be a mismatch even if the first characters match the user search pattern. So the cloud returns also B'[pos]. The client runs the permutation $\text{pos}' = \Pi_{\text{k}_{\pi_3}}(\text{pos})$ and asks the cloud to return the bucket $\text{b}_{pos'}$. Then client checks if the first m mod win characters of the decrypred bucket equals the last m mod win characters of the search pattern and accepts the pos as valid otherwise discards the result.

7 SECURITY ANALYSIS

We illustrate the security of the scheme pertaining to definition 2.2. User splits the original data stream S in consecutive buckets of size win by moving each time the window to the right one character. More specifically, we show the existence of a simulator $\mathcal{S}$, who has access to the leakage function $\mathcal{L}$ and produces indistinguishable views to an adversary $\mathcal{A}$. Conceptually the proof demonstrates that an adversary $\mathcal{A}$, who can be a semi-honest cloud cannot learn more information from what it can be leaked in an ideal work without malicious behaviors. During the encryption of the index users write to the LLSet linked list and to a three dimensional array FM. FM consists of three columns: F, L from the BWT transformation, which are used to traverse the string, and the SA column, which consists of encrypted positions of the corresponding suffixes. To avoid leakage of frequencies users pad each linked list LLSet. In order to impede off-line traversal of the index, the user further encrypts the header of each linked list. Moreover, two different permutations $\Pi_{\text{k}_{\pi_{1,2}}}$ are applied first on all the rows of the first two columns F, L and then to the SA column to prevent an adversary from correlating encrypted positions to intermediate results after having obtained a token. I.e: without the permutation, $\mathcal{A}$ after obtaining a token can stop the search to a substring of the queried substring and learn the associated encrypted position from SA column. After $\mathcal{A}$ observing LLSet and FM it learns the size of the padded string n'.

During the the search phase the adversary $\mathcal{A}$ observes encrypted tokens. Between two encrypted tokens $\mathcal{A}$ can identify similarity in between two tokens: i.e: whether two identical token have been issued, or identical encrypted buckets in between the tokens. Then we point out two different cases:

(1) **Size of token multiple of** win: The user leaks to the cloud how many matches exist in the string S for the queried token, the encrypted cells at SA, which correspond to the matched positions.

(2) **Size of token not a multiple of** win: The user in case of a token, whose size is not a multiple of the size win of each bucket, needs to verify the correctness of the potential encrypted match(s). As such the cloud returns the permuted possible position pos. User decrypts and applies $\text{pos}' = \Pi_{\text{k}_{\pi_3}}(\text{pos})$ and asks the cloud to return the bucket at position pos'. The cloud learns the encrypted position of the last bucket of the token and its encrypted text.

Integrity guarantees of the data and the index are assured thanks to the use of authenticated symmetric cryptography. Before stating our theorem concerning the security of S^3E we formally define our leakage function $\mathcal{L}$ from our aforementioned analysis.

Definition 7.1. A leakage function $\mathcal{L}$ for a S^3E scheme comprises the following three leakage functions:

- PreProcess Leakage: $\mathcal{L}_1$ includes the padded size of the data stream $n' >= |S|$.
- SrchToken Leakage: The SrchToken Leakage $\mathcal{L}_2$ reveals the length of the token |tk|, how many common characters reside in it and similarity patterns between different tokens.
- Search Leakage: $\mathcal{L}_3$ leaks how many times a substring token tk exists in the padded string S with dummy blocks.
- Intermediate: Leakage $\mathcal{L}_4$ leaks intermediate addresses of the F and L columns of a query.
- Access pattern: Leakage $\mathcal{L}_5$ leaks the addresses of the SA cells for a fixed query.

To eliminate $\mathcal{L}_2, \ldots, \mathcal{L}_5$ leakage profiles an expensive ORAM scheme can be used, however it is out of the scope of the paper, as it will increase communication and storage overhead drastically.

THEOREM 7.2. *Let* $\mathsf{F}_{\mathsf{k_f}}, \Pi_{\mathsf{k}_\pi}, \mathsf{SKE} = \{\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}\}$ *be a pseudorandom function, a pseudorandom permutation, a semantically secure symmetric encryption scheme respectively, then our substring searchable symmetric encryption scheme* $\mathsf{S^3E}$ *is adaptively* $\mathcal{L}$*-semantically secure.*

Game	Change	Indistinguishability Argument
Game_0	$\mathsf{Game}_0 = \mathbf{Real}^{\mathsf{S^3E}}_{\mathcal{A}(\lambda)}$	By definition
Game_1	Replace $\mathsf{F}_{\mathsf{k_f}}$, $\mathsf{F}_{\mathsf{k_l}}$	Pseudorandomness of $\mathsf{F}_{\mathsf{k_f}}$
Game_2	Replace $\Pi_{\mathsf{k}_{\pi_{1,2,3}}}$,	Pseudorandomness of Π_{k_π}
Game_3	Replace SKE = {Gen, Enc, Dec}	Semantically secure SKE = {Gen, Enc, Dec}
Game_4	$\mathsf{Game4} = \mathbf{Ideal}^{\mathsf{S^3E}}_{\mathcal{A},\mathcal{S}(\lambda)}$	By definition

Table 2: Hybrid games

THEOREM 7.3. *Let* $\mathsf{F}_{\mathsf{k_f}}, \Pi_{\mathsf{k}_\pi}, \mathsf{SKE} = \{\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}\}$ *be a pseudorandom function, a pseudorandom permutation and a semantically secure symmetric encryption scheme respectively, then our substring searchable symmetric encryption scheme* $\mathsf{S^3E}$ *is adaptively* $\mathcal{L}$*-semantically secure.*

PROOF. (Sketch)

In the $\mathbf{Real}^{\mathsf{S^3E}}_{\mathcal{A}(\lambda)}$ world (cf. 2.2) the adversary can obtain the encrypted index and encrypted keywords of its choice. In the beginning the Challenger selects the size of the buckets win and uniformly at random keys $\mathsf{k} = (\mathsf{k_f}, \mathsf{k_l}, \mathsf{k}_{\pi_{1,2,3}}, \mathsf{k_{e,c}})$ for a PRF $\mathsf{F_k} : \{0,1\}^\lambda \times \{0,1\}^\nu \to \{0,1\}^\mu$, a PRP $\Pi_{\mathsf{k}_\pi} : \{0,1\}^\lambda \times \{0,1\}^\nu \to \{0,1\}^\nu$ and a symmetric encryption algorithm $\mathsf{SKE} = \{\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}\}$. Upon receipt of a stream S of size n, the Challenger employs the $\mathsf{SES} \leftarrow \mathsf{PreProcess}(\mathsf{k}, S)$ as presented in section 6 and forwards SES to $\mathcal{A}$. We distinguish between matching q^m and non-matching queries $q^{nm} : \mathbf{q} = \bigcup q^{nm} q^m$. We assume for the ease of readability that adversary issues only matching queries q^m. During the Search phase, C has also access to $\mathcal{L}_2$=similarities between tokens, $\mathcal{L}_3$=# times a token exist in the substring. C upon receipt of $\mathbf{q}$ checks in a table QT whether $\mathbf{q} \in QT$. If so C fetches the corresponding token $\mathsf{tk}_{T,S}$ and forwards it to $\mathcal{A}$. If this is the frist time for $\mathbf{q}$ then C computes $\mathsf{tk}_{T,S} = C[1], C[2], \cdots, C[m] \leftarrow \mathsf{F}_{\mathsf{k_f}}(T[1...m]), \mathsf{F}_r(T[m])$. Finally $\mathcal{A}$ receives $\mathbf{t} = (\mathsf{tk}_1, \mathsf{tk}_2, \mathsf{tk}_3, \ldots, \mathsf{tk}_o)$ for each substring query.

Within a sequence of hybrid games we show the indistinguishable transformation of $\mathbf{Real}^{\mathsf{S^3E}}_{\mathcal{A}(\lambda)}$ game to eventually the $\mathbf{Ideal}^{\mathsf{S^3E}}_{\mathcal{A},\mathcal{S}(\lambda)}$ game, which concludes the proof. The simulator $\mathcal{S}$ computes the simulated encrypted index $\mathsf{SES}^* = (\mathsf{LLSet}^*, \mathsf{LL}^*_c, \mathsf{FM}'^*)$ as follows:

- Game_0: This game is equivalent with the $\mathbf{Real}^{\mathsf{S^3E}}_{\mathcal{A}(\lambda)}$ game.
- Game_1: This game behaves as the $\mathbf{Real}^{\mathsf{S^3E}}_{\mathcal{A}(\lambda)}$ game with the difference that $\mathcal{S}$ does not have access to S. The simulator through the $\mathcal{L}_1$ leakage function builds the substring encrypted structure SES as follows: We assume the existence of an algorithm $S \leftarrow \mathsf{Build}(n', str)$, which takes as input $n' \in \mathbb{N}$ and the structure $str = \{\mathsf{b}\}_{i=1}^{n'}, \mathsf{b} \in \Sigma^*$ and outputs a bitstring of length n', from a vocabulary Σ^*. Notice that as in the real game the valid length of the original stream is not revealed and only the length of the string after the padding n' is leaked. $\mathcal{S}$ selects uniformly at random keys $\mathsf{k} = (\mathsf{k_f}, r, \mathsf{k}_\pi, \mathsf{k_e})$ for a PRF $\mathsf{F}_{\mathsf{k_f}} : \{0,1\}^\lambda \times \{0,1\}^\nu \to \{0,1\}^\mu$, a PRP $\Pi_{\mathsf{k}_\pi} : \{0,1\}^\lambda \times \{0,1\}^\nu \to \{0,1\}^\nu$ and a symmetric encryption algorithm $\mathsf{SKE} = \{\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}\}$ and runs $\mathsf{SES} \leftarrow \mathsf{PreProcess}(\mathsf{k}, \mathsf{Build}(\mathcal{L}_1))$. $\mathcal{S}$ uses $\mathsf{F}_{\mathsf{k_f}}$ to evaluate bit strings of length c_n: $\mathcal{L}_2(q) = c_n$. For the simulation of the tokens and its responses, $\mathcal{S}$ uses the leakage obtained from $\mathcal{L}_2 \ldots \mathcal{L}_5$.
- Game_2: This game behaves similarly with Game_1, but we replace the $\mathsf{F}_{\mathsf{k_f}}$ with a real random function which is evaluated through access to an oracle $O^{\mathsf{RF}}(\lambda, \mu, \nu)$.
- Game_3: This game behaves similarly with Game_2, but we replace the Π_{k_π} with a real random permutation which is evaluated through access to an oracle $O^{\mathsf{R}\Pi}(\lambda, \nu)$.
- Game_4: In Game_4 we replace the semantically secure SKE = {Gen, Enc, Dec} with real random values by querying an oracle $O^{\mathsf{RE}}(\lambda)$.

We write $\mathsf{Game}_i \approx \mathsf{Game}_j$ to denote that the view of probabilistic polynomial time adversary $\mathcal{A}$ is indistinguishable between the output of Game_i and Game_j. $\mathsf{Game}_0 = \mathbf{Real}^{\mathsf{S^3E}}_{\mathcal{A}(\lambda)}$ by definition, $\mathsf{Game}_1 \approx \mathsf{Game}_0$ as long as no collisions happen to the evaluation of $\mathsf{F}_{\mathsf{k_f}}$, Π_{k_π}, SKE = {Gen, Enc, Dec} or E, $\mathsf{Game}_2 \approx \mathsf{Game}_1$ as long as $\mathsf{F}_{\mathsf{k_f}}$ is indistinguishable from real random function, $\mathsf{Game}_3 \approx \mathsf{Game}_2$ thanks to the indistinguishable output of Π_{k_π} from real random permutations, $\mathsf{Game}_4 \approx \mathsf{Game}_3$ because of the semantically secure SKE = {Gen, Enc, Dec} and finally $\mathsf{Game}_5 = \mathbf{Ideal}^{\mathsf{S^3E}}_{\mathcal{A},\mathcal{S}(\lambda)}$ by definition.

□

8 PERFORMANCE

In this section we present our implementation results. We demonstrate the practicality of $\mathsf{S^3E}$ with benchmark experiments, comparing our results with the scheme of Chase *et al.* [7], in order to validate the claims of our performance improvements. To accomplish the comparison we also implemented the suffix tree based construction of [7] called hereafter ST.

8.1 Implementation

Our comparison is based on two metrics: **a) storage overhead** for the encryption of the index and the computation of a substring query as an encrypted token, **b) computation time** of each operation. The reported computation times for each experiment are taken as the average of 100 trials. As we implemented both schemes on the same machine, which simulates both the client and the server, we can derive accurate and fair observations about the performance of the protocols on real metrics. In the real world the server can be implemented in a more powerful machine, however this does not change the storage overhead or the computation performance fraction of both schemes.

Thanks to our encrypted suffix array construction, we achieve a storage improvement by a factor of **1.71** for the DNA sequence data stream and **1.57** for the enron email data. This occurs first

because of the extra information a node of the suffix tree should keep (leaf nodes, parent nodes, auxiliary information for the substring of the path in the tree) and due to the dummy nodes policy, which increases the size of the tree. Subsequently that affects the computation time for the computation of the encrypted index with a $\approx$ **4x** blowup on average for data sets of size 10^6.

8.2 Benchmarks

8.2.1 Index. In tables 3,4, we depict the storage and computational overhead incurred by the computation of the encrypted index using the suffix array in S^3E using different window sizes for the buckets and the suffix tree in ST scheme of [7].

We observed an increased overhead in the size of the encrypted index for the ST scheme [7], compared with ours as expected. On average, over all the the data sizes, for different data sets the gain of S^3E over ST approximates a factor of **1.64**. However, for realistic big data streams consisting of 10^6 the gain reaches a factor of **2**. Even though the computation of the encrypted index happens only once, the storage overhead incurred by its encryption is of more crucial importance than its computation time. A limited storage device is not capable of computing the encrypted index if that comes at an increased communication overhead.

We also measured the computation time of the encrypted index in both schemes in tables 3, 4. The S^3E index construction time outperforms ST. Apart from the extra dummy blocks and the increased size of the suffix tree compared with that of a suffix array, the increased computation cost stems from the way ST encrypts the suffix tree.

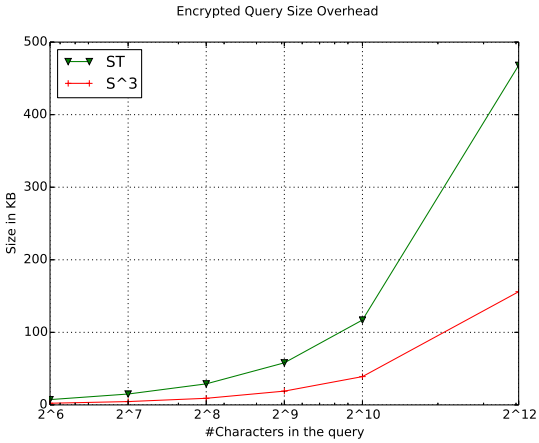

Figure 3: Token storage overhead for both schemes in DNA streams. As the window size affects very epidemically the encryption of our query we use a fixed window size: win = 2^5

8.2.2 Query Encryption. We run experiments in order to compute the storage overhead during the SrchToken phase. The token consists of a sequence of buckets from a vocabulary: be it characters from emails or characters from DNA sequence. As the query encryption is not affected by the distribution of the underlying characters and for compactness we choose to present results only from the DNA sequence. We also observed tiny differences at the

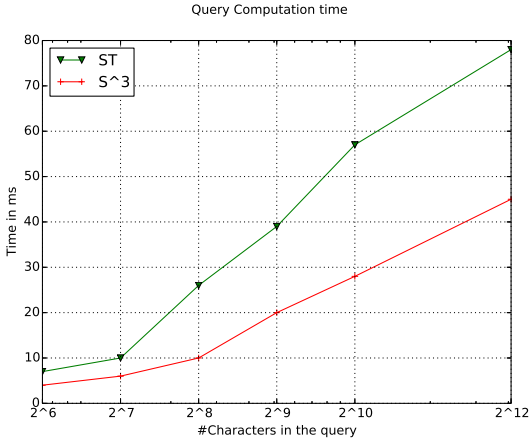

Figure 4: Token computation time for both schemes in DNA streams. As the window size affects very epidemic the encryption of our query we use a fixed window size: win = 2^5

query encryption time and the size of the bucket. We observe a reasonable increase in both the size of the encrypted query (figures 3,4) and its computation time as the query size increases. S^3E outmatches in query computation time due to the way the query is encrypted in ST scheme: Namely for each character 2 PRF, and one block cipher is invoked, while in S^3E only one PRF is invoked. The storage overhead of ST also outgrows faster since the substring is encrypted recursively and not by character. That is, the token: $T[1], T[2], ..., T[m]$ is encrypted as $ct_1 = PRF_1(T[1]), k_1 = PRF_2(T[1]), ct_2 = PRF_1(T[1]T[2]), k_2 = PRF_2(T[1]T[2])$ and so on. Finally the client forwards to the cloud: $\{Enc_{k_i}(ct_i)\}_{i=1}^{m}$.

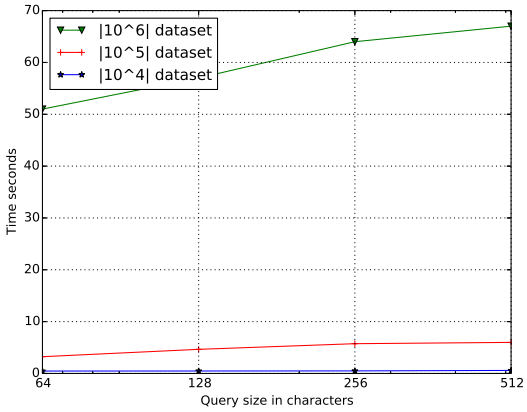

Figure 5: Response time for S^3E scheme in a DNA stream. Time is measured as the average over different bucket sizes: 2, 2^3, 2^5

8.2.3 Response Overhead. In figures 5, 6 we discern a slight outperformance of S^3E compared with ST in terms of substring response time. For the experiments we computed tokens of various lengths and perform a search on DNA streams of different sizes.

Dataset	#Characters									
	10^2		10^3		10^4		10^5		10^6	
	Storage	**Time**	**Storage**	**Time**	**Storage**	**Time**	**Storage**	**Time**	**Storage**	**Time**
DNA	60KB	0.44s	567KB	1.07s	6.4MB	9.66s	63MB	89.50s	589MB	1382s
Enron	57KB	0.42s	562KB	0.97s	6.3MB	9.07s	62MB	85.74s	579MB	1268s

Table 3: [7] Index storage and computational overhead for variable size data sets.

Dataset	#Characters									
	10^2		10^3		10^4		10^5		10^6	
	Storage	**Time**	**Storage**	**Time**	**Storage**	**Time**	**Storage**	**Time**	**Storage**	**Time**
DNA										
win = 2	37KB	0.26s	410KB	49s	4.2MB	3.50s	42MB	29.90s	456MB	728s
win = 2^3	38KB	0.15s	401KB	48s	4.1MB	3.46s	40MB	29.78s	410MB	683s
win = 2^5	29KB	0.11s	371KB	43s	3.7MB	2.34s	37MB	26.12s	373MB	640s
Enron										
win = 2	40KB	0.29s	402KB	50s	4.1MB	3.69s	41MB	30.65s	471MB	702s
win = 2^3	38KB	0.17s	396KB	46s	4MB	3.42s	40MB	29.23s	401MB	678s
win = 2^5	29KB	0.10s	385KB	42s	3.7MB	2.56s	37MB	27.50s	372MB	630s

Table 4: S^3E Index storage and computational overhead for variable size data sets and buckets window size.

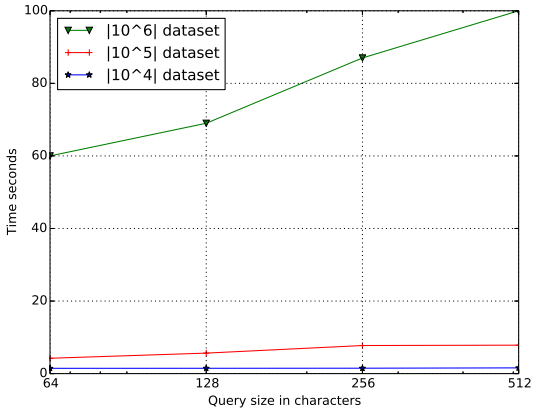

Figure 6: Response time for ST [7] scheme

The client computes and encrypts the index and uploads it to the cloud. The cloud simulated in the same machine runs the search algorithm, and we computed the total search time. We perceived in both schemes, that for considerable smaller than 10^6 elements the running time tends to be independent on the size of the substring token. For a one million data stream there is a notable increased response time compared with the smaller data sets and there is a proportional increment in time with respect to the size of the token. In exact times, S^3E surmounts ST [7] for the computation of the response at the cloud side. This outperformance is due to the increased size of the encrypted index in [7] with dummy blocks, based on a suffix tree data structure.

9 COMPARISON

We perform a comparison of our S^3E with existing solutions (cf. table 5). We analyzed the search running time in asymptotic complexity, index space requirements both in the plaintext and in the ciphertext space, query size, variable length capability, rounds of communication and search leakage. Since our scheme competes mostly with [7] here we further elaborate its cost analysis from table 5.

Search. Thanks to the usage of encrypted dictionary the cost of searching a m length string is $O(m+k)$, where k denotes the number of occurrences. However, due to the extra dummy blocks the search cost is increased to $O(m + k + \sum_{j=1}^{k} (\max_b - f_{b_i}))$, where $\max_b$ is the most frequent bucket and f_{b_i} the frequency of bucket b_i.

Index. For the index space complexity, we analyzed the space requirement in the plaintext space and in the ciphertext space. For the plaintext space analysis we assume, that a pointer or integer requires 4 bytes. Recall that a suffix tree has n leaves, at most $n-1$ internal nodes and at most $2n-2$ edges. Thus, for a naive suffix tree implementation we need 2 pointers for each leaf: one for the parent node and one for its position to the original stream, resulting in $8n$ bytes. Four pointers for each internal node: one for the parent node, one for each leftmost child, one for the right sibling and one pointer for the suffix link, which reduces the search time during a substring query. The total storage cost for the internal nodes is $4 * 4n = 16n$. For each edge, suffix trees allocate one pointer for the beginning position of the substring in the stream and one for the end position of the substring in the stream increasing the space cost to $24n + 4 * 2 * 2n = 40n$. The space cost of the solution based on suffix trees [7] can be further reduced to $20n$ by eliminating the need to store suffix links and parent pointers. However, the extra dummy blocks further augment the storage overhead. Assuming a suffix tree with N internal, each node is further padded with dummy children nodes so as to each node has σ children, where σ is the size of the vocabulary. Furthermore, the internal nodes are padded with up to $2n-2$ nodes where n is the size of the string. Finally the size of the extra dummy nodes in [7] is: $\Sigma_{i=1}^{2n-(2+N)}(\sigma - child(i))$, where $child(i)$ equals the number of children for internal node i in the suffix tree. In contrast, in S^3E we replace the space expensive suffix trees with suffix arrays and as such the index space cost is reduced from $20n$ bytes to $4n$ bytes.

For the storage space computation during the encryption of the index, be it suffix tree or suffix array, we exclude a per byte comparison and we assume a ciphertext comparison. The encryption of the index is based on the translation of the suffix tree to an encrypted dictionary. Thus, all the extra pointers of the suffix tree

Protocol	Search	Index [PS|CS]		Query [FR|LR]		VLS	Rounds	SL
CS[7]	$O(m+k)$	$20n$	$4(n+\sum_{i=1}^{2n-(2+N)}\sigma - child(i))$	$\frac{m(m+1)}{2\theta}$	m+k	✓	3	SP+QPP+IIP+LIP
FJKNRS[11]	$O(n)$	-		m	0	✓	1	SP
FHV[12]	$O(n-m)$	-		m	0	✗	1	✗
S^3E	$O(m+k)$	$4n$	$4(n+\sum_{j=1}^{k}\max_b - f_{b_i})$	m	1	✓	2, 3	SP+QPP+IIP

Table 5: Comparison of existing substring searchable encryption protocols. Index space is further categorized in plaintext space index storage space (PS) and ciphertext space (CS). The overhead of [11] and [12] is undefined as the schemes do not take advantage of any auxiliary data structure for efficient substring search. For the query complexity we analyzed its size in terms of two separated phases: at the first round (FR) of the protocol and the last one (LR), in case of multiple rounds protocols. VLS denotes variable length substring search and SP the search patternleakage: QPP: Query prefix pattern, IIP: Index intersection pattern, LIP: Leaf intersection pattern.

are excluded. Following the protocol from [7], the user encrypts $2n$ substrings which are equal to the number of edges of the suffix tree plus n leaves and n characters of the original stream, resulting in $4n$ encryptions. In our solution thanks to the FM mapping the user sends the encrypted suffix array, plus two more n size arrays for the FM index construction; one for the F column of the index and one for the L column. In the end it uploads $4(n+\sum_{j=1}^{k}\max_b - f_{b_i})$ encrypted values to the cloud, in total.

Query. For the query size, we assume a block cipher of size θ and a substring query of size m. In [7] the substring is encrypted incrementally: for the substring "abc" user encrypts separately E(a), E(ab), E(abc). As such, for big substring queries as in DNA queries, the number of ciphertexts exceeds the number of the substring m. The total number of encryptions equals $\frac{1}{\theta}+\frac{2}{\theta}+\cdots+\frac{m}{\theta}=\frac{m(m+1)}{2\theta}$ during the first round. At the last round the user asks for the positions of each character separately augmented by a factor of m the substring size. In S^3E the substring query has only per character encryptions of each character in the first round plus a ciphertext for the last round. Our solutions also allows variable size substring queries, since the size of the substring query is decoupled from the scheme and can be defined online during the query phase as in [7].

Rounds. Regarding the rounds of communication, S^3E can return the substring search results in 2 rounds of communication or in 3 when the query size is not a multiple of the bucket size win. During the first round, the client sends an encrypted substring query and the cloud responds with the encrypted addresses of the corresponding suffix array positions. At the second round the client decrypts the permuted position of the suffix tree and asks the cloud for the unpermuted encrypted position. A third round is performed when a query is not a multiple of the size of the bucket, whereby the client verifies the correctness of the possible match.

Security Leakage. Concerning the search leakage, Chase *et al.* [7] scheme leaks the search pattern, meaning an untrusted could can identify similarities between two or more substring search queries. Moreover, the scheme reveals the query prefix pattern, which leaks whether a node has been visited for a previous substring in the suffix tree, the index intersection pattern which allows the cloud to learn if the returned index position has already been asked and finally, the leaf index leaks when any of the returned positions of the tree leaves have been previously queried. Since in S^3E we avoid the use of a suffix tree, S^3E does not leak the leaf pattern. We inherit though, the index intersection pattern, which reveals if any returned index has been returned in a previous query. As in [7] our scheme reveals also the cloud differences of the indexes when a user asks for substrings that they do differ in one position and there is only a single position in the original stream S. Both schemes employ a padding policy to add dummy blocks in order to obfuscate the structure of the index and the stream. S^3E is also adaptively secure under the real-ideal simulation paradigm. We also use an authenticated encryption scheme in order to assure the integrity of the messages.

10 CONCLUSION

We designed and analyzed a substring searchable symmetric encryption protocol S^3E, which achieves better storage performance than state of the art work [7]. The idea of our protocol is to leverage the self-indexing mechanism of FM index, which stores only n integer positions of its substrings. Our protocol is provably secure under the real-ideal paradigm. We also implemented our protocol and compared it with the state of the art work [7], showing its notable performance improvement in terms of storage overhead and computation time.

As part of future we will investigate solutions for substring queries on encrypted data, which further reduce the search time by tolerating accuracy. This is achieved with approximation algorithms. To the best of our knowledge, at the time of this writing, the literature has not addressed how to perform approximation queries for substring queries on encrypted data. Moreover, we seek to look for solutions, which scale better in terms of computational time by leveraging parallelization: Due to the large amount of data produced in the stream of a DNA sequence, the client may outsource the task of computing the FM index in different servers, each one holding a portion of the data stream. It is becoming challenging how those servers can compute the encrypted index. Recent progress in the area of algorithms has shed some light [36], but tweaking the algorithms to operate on encrypted data without leaking more than it is accepted, needs further research.

REFERENCES

[1] Mohamed Ibrahim Abouelhoda, Enno Ohlebusch, and Stefan Kurtz. 2002. Optimal Exact String Matching Based on Suffix Arrays. In *In Proceedings of the Ninth International Symposium on String Processing and Information Retrieval. Springer-Verlag, Lecture Notes in Computer Science.*

[2] Robert S. Boyer and J. Strother Moore. 1977. A Fast String Searching Algorithm. *Commun. ACM* 20, 10 (Oct. 1977), 762–772.

[3] M. Burrows and D. J. Wheeler. 1994. *A block-sorting lossless data compression algorithm.* Technical Report.

[4] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2013. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I.* 353–373.

[5] Yan-Cheng Chang and Michael Mitzenmacher. 2005. Privacy Preserving Keyword Searches on Remote Encrypted Data. In *Proceedings of the Third International Conference on Applied Cryptography and Network Security (ACNS'05)*. Springer-Verlag, Berlin, Heidelberg, 442–455.

[6] Melissa Chase and Seny Kamara. 2010. Structured Encryption and Controlled Disclosure. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*. 577–594.

[7] Melissa Chase and Emily Shen. 2015. Substring-Searchable Symmetric Encryption. *PoPETs* 2015, 2 (2015), 263–281. http://www.degruyter.com/view/j/popets.2015.2015.issue-2/popets-2015-0014/popets-2015-0014.xml

[8] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*. ACM, New York, NY, USA, 79–88.

[9] Emiliano De Cristofaro, Sky Faber, and Gene Tsudik. 2013. Secure Genomic Testing with Size- and Position-hiding Private Substring Matching. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society (WPES '13)*. ACM, New York, NY, USA, 107–118.

[10] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. 2015. Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM. Cryptology ePrint Archive, Report 2015/005. (2015). http://eprint.iacr.org/2015/005.

[11] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel-Catalin Rosu, and Michael Steiner. 2015. Rich Queries on Encrypted Data: Beyond Exact Matches. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II*. 123–145.

[12] Sebastian Faust, Carmit Hazay, and Daniele Venturi. 2013. Outsourced Pattern Matching. In *Automata, Languages, and Programming*, FedorV. Fomin, Rsiš Freivalds, Marta Kwiatkowska, and David Peleg (Eds.). Lecture Notes in Computer Science, Vol. 7966. Springer Berlin Heidelberg, 545–556.

[13] P. Ferragina and G. Manzini. 2000. Opportunistic Data Structures with Applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS '00)*. IEEE Computer Society, Washington, DC, USA, 390–. http://dl.acm.org/citation.cfm?id=795666.796543

[14] Rosario Gennaro, Carmit Hazay, and Jeffrey S. Sorensen. 2016. Automata Evaluation and Text Search Protocols with Simulation-Based Security. *J. Cryptology* 29, 2 (2016), 243–282.

[15] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. 2013. Optimizing ORAM and Using It Efficiently for Secure Computation. In *Privacy Enhancing Technologies - 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings*. 1–18.

[16] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (May 1996), 431–473.

[17] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. 1992. Information Retrieval. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, Chapter New Indices for Text: PAT Trees and PAT Arrays, 66–82.

[18] Carmit Hazay and Yehuda Lindell. 2010. Efficient Protocols for Set Intersection and Pattern Matching with Security Against Malicious and Covert Adversaries. *J. Cryptology* 23, 3 (2010), 422–456.

[19] Carmit Hazay and Tomas Toft. 2010. Computationally secure pattern matching in the presence of malicious adversaries. In *Advances in Cryptology-ASIACRYPT 2010*. Springer Berlin Heidelberg, 195–212.

[20] Juha Kärkkäinen and Peter Sanders. 2003. Simple Linear Work Suffix Array Construction. In *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*. 943–955.

[21] Richard M. Karp and Michael O. Rabin. 1987. Efficient Randomized Pattern-matching Algorithms. *IBM J. Res. Dev.* 31, 2 (March 1987), 249–260.

[22] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. 1977. Fast Pattern Matching in Strings. *SIAM J. Comput.* 6, 2 (1977), 323–350.

[23] Yehuda Lindell. 2016. How To Simulate It - A Tutorial on the Simulation Proof Technique. *IACR Cryptology ePrint Archive* 2016 (2016), 46. http://eprint.iacr.org/2016/046

[24] Udi Manber and Gene Myers. 1990. Suffix Arrays: A New Method for On-line String Searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '90)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 319–327.

[25] Edward M. McCreight. 1976. A Space-Economical Suffix Tree Construction Algorithm. *J. ACM* 23, 2 (April 1976), 262–272.

[26] Tarik Moataz and Erik-Oliver Blass. 2015. Oblivious Substring Search with Updates. Cryptology ePrint Archive, Report 2015/722. (2015). http://eprint.iacr.org/2015/722.

[27] Tarik Moataz, Travis Mayberry, and Erik-Oliver Blass. 2015. Constant Communication ORAM with Small Blocksize. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 862–873.

[28] Dimitrios Papadopoulos, Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. 2015. Practical Authenticated Pattern Matching with Optimal Proof Size. *Proc. VLDB Endow.* 8, 7 (2015), 750–761.

[29] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. 2014. Blind seer: A scalable private dbms. In *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 359–374.

[30] Olivier Sanders, Cristina Onete, and Pierre-Alain Fouque. 2017. Pattern Matching on Encrypted Streams: Applications to DPI and searches on genomic data. Cryptology ePrint Archive, Report 2017/148. (2017). http://eprint.iacr.org/2017/148.

[31] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with O ((logN) 3) worst-case cost. In *Advances in Cryptology–ASIACRYPT 2011*. Springer Berlin Heidelberg, 197–214.

[32] E. Stefanov and E. Shi. 2013. ObliviStore: High Performance Oblivious Cloud Storage. In *Security and Privacy (SP), 2013 IEEE Symposium on*. 253–267.

[33] Emil Stefanov, Elaine Shi, and Dawn Song. 2011. Towards practical oblivious RAM. *arXiv preprint arXiv:1106.3652* (2011).

[34] E. Ukkonen. 1958. On-line construction of suffix trees. *Algorithmica* 14, 3 (1958), 249–260.

[35] Peter Weiner. 1973. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT '08. IEEE Conference Record of 14th Annual Symposium on*. 1–11.

[36] Liu Y., Hankeln T., and Schmidt B. 2016. Parallel and Space-Efficient Construction of Burrows-Wheeler Transform and Suffix Array for Big Genome Data. *IEEE/ACM Trans Comput Biol Bioinform* 3 (2016), 592–598.